MGTS 04
final version

Extracting Information from Structured Documents with Automata in a Single Run

Stefan Raeymaekers and Maurice Bruynooghe

K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Leuven,
{stefanr,maurice}@cs.kuleuven.ac.be

Abstract. A proved approach in information extraction from documents is to learn an automaton from a number of examples where one of the fields to be extracted is marked with a special symbol. Then the actual extraction marks in turn each candidate field and checks whether the automaton accepts the document. This requires as many runs of the automaton as there are candidate fields. In this paper we develop a method where all accepted fields are extracted in a single run and show that this results in a substantial speed-up of the extraction phase. A major difference is that we use an automaton that accepts only documents in which every element that should be extracted is marked, instead of an automaton that accepts documents in which it suffices that some of these elements are marked.

1 Introduction

The aim of information extraction(IE) systems is to extract specific information from a set of human readable documents into a form suitable for computer manipulation. Two types of IE can be distinguished. This paper focuses on IE from semi-structured documents, like web pages and XML documents. The other type extracts information from unstructured documents (meaning texts), using techniques borrowed from natural language processing. Techniques for IE from semi-structured documents make use of *wrappers*, customized extraction procedures that take the structure of the document into account. Several techniques to automatically learn wrappers have been proposed. Some examples can be found in [7, 8, 12, 2, 5, 6]. We are interested in the approach presented in [5], because it needs only very few examples to learn, the learning process is fast, and it results in highly accurate wrappers, due to the exploitation of the tree structure of the documents. A disadvantage is that the extraction phase is very slow.

In [5] tree automata are used to accept or reject markings of a document. Elements that are candidates for extraction are marked with a special symbol. A *correct marking* (for a given document and extraction task) is a marked version of the document where an element is marked iff it has to be extracted. Using different symbols as markers, a correct marking can address several extraction tasks (assuming the same element need not be extracted for different tasks). An automaton that accepts correctly marked documents and rejects all others is a *correct marking acceptor*(CMA). A *partially correct marking* is a marked version

of the document where the marked elements are a subset of the elements marked in the correct marking. An automaton that accepts partially correctly marked documents is a *partially correct marking acceptor*(PCMA).

In [5,6], a PCMA is learned from partially correct marked examples where *one* element is marked. When used to extract information from a document with n targets, n runs are performed, where one candidate target is marked in each run; the element is extracted when the automaton accepts the partially marked document. As a result, the extraction time is quadratic in the size of the documents.

We developed a method to convert a PCMA in a CMA [9]. In the current paper, we describe how to extract all information in a single run. This results in a substantial speed-up of the extraction task. The method is based on the availability of a CMA for the extraction task. In a first step we describe how a CMA can be extended so that it not only accepts or rejects a document, but also outputs all marked elements. In a second step we show how it can be further modified to consider all candidate markings in parallel (without building them explicitly.) As a result, all elements in the correct marking of the document can be extracted in a single run. While the complexity of such a run is higher than the complexity of a single run during extraction with a PCMA, overall, there is a substantial reduction of extraction time.

We introduce these techniques not only for tree automata, but also for string automata. On one hand because understanding the string case helps to understand the more general tree case; on the other hand because extraction time is also reduced in case of string automata.

We provide some preliminary definitions in Section 2. The method is described in Section 3. In Section 4 we discuss the complexity of our method; some experimental results are given in Section 5. In Section 6, we argue that the approach is also suited for more general extraction tasks than extraction of leaf elements in document trees, and can be used to count structured patterns. We finally conclude in Section 7.

2 Preliminary Definitions

Since we handle in this paper both string and tree documents, we overload a lot of definitions and functions. Where necessary, we will distinguish between them.

2.1 Marked Documents

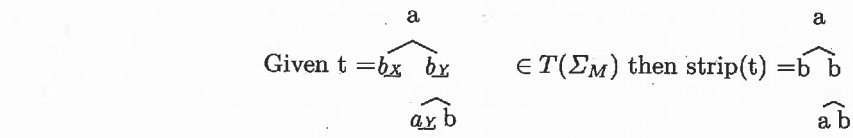
We define the alphabet Σ as a finite set of symbols. The set of all strings over Σ is denoted as Σ^* while the set of all unranked trees over Σ is inductively defined as $T(\Sigma) = \{f(s) \mid f \in \Sigma, s \in T(\Sigma)^*\}$. We usually denote $f(\epsilon)$, where ϵ is the empty sequence, by f. In what follows, we will use $doc(\Sigma)$ to denote either Σ^* or $T(\Sigma)$.

Let M be the set of markers that we consider in the extraction task. A marked alphabet Σ_M associated to a given alphabet Σ and the set of markers M is then

defined as: $\Sigma_M = \Sigma \cup \{e_{\underline{X}} \mid e \in \Sigma, X \in M\}$. We call a symbol $e \in \Sigma$, an *unmarked symbol* and $e_{\underline{X}} \in \Sigma_M$, a *marked symbol.* The function *strip*: $\Sigma_M \rightarrow \Sigma$ is defined as: strip(e) = e for an unmarked symbol and as $strip(e_{\underline{X}}) = e$ for a marked symbol. We overload the strip function and use it also as a function from $doc(\Sigma_M)$ to $doc(\Sigma)$, to obtain an unmarked copy of a document.

The set of all markings of an unmarked symbol or document d is defined as: $mark_M(d) = \{d_M \in S_M \mid strip(d_M) = d\}$, where S_M is respectively Σ_M, Σ_M^* and $T(\Sigma_M)$ for a symbol, a string and a tree.

Example 1. Given the alphabet $\Sigma = \{a, b\}$ and the set of markers $M = \{X, Y\}$. We have $\Sigma_M = \{a, a_{\underline{X}}, a_{\underline{Y}}, b, b_{\underline{X}}, b_{\underline{Y}}\}$, $mark_M(a) = \{a, a_{\underline{X}}, a_{\underline{Y}}\}$, and $mark_M(ba) = \{ba, ba_{\underline{X}}, ba_{\underline{Y}}, b_{\underline{X}}a, b_{\underline{X}}a_{\underline{X}}, b_{\underline{X}}a_{\underline{Y}}, b_{\underline{Y}}a, b_{\underline{Y}}a_{\underline{X}}, b_{\underline{Y}}a_{\underline{Y}}\}$, where $ba \in \Sigma^*$.



2.2 Sequence automata and tree automata

A non-deterministic Finite String (or Sequence) Automaton (NFSA) is a tuple $\mathcal{A} = (\Sigma_i, \Sigma_o, Q, q_0, \delta, \phi)$ where Σ_i is a set of input symbols, Σ_o is a set of output symbols, Q is a set of states, $q_0 \in Q$ is the initial state, δ is a transition relation from the pairs (current state, input symbol) to the next states ($\delta \subseteq (Q \times \Sigma_i) \times Q$), and ϕ is an output function $Q \rightarrow \Sigma_o$. A finite string automaton is deterministic (DFSA) if the transition relation is a function. Unless otherwise indicated we will assume that an FSA is deterministic. We extend the transition function to a function $\hat{\delta} : (Q \times \Sigma_i^*) \rightarrow Q$, that is defined as $\hat{\delta}(q, \epsilon) = q$, and $\forall s \in \Sigma_i^*$ and $a \in \Sigma_i$, $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$. An FSA with Σ_o={accept, reject} is called a Finite String Acceptor.

Example 2. As documents we consider strings that consist of zero or more sequences of the pairs of symbols @+, @−, and @?. Let the set of markers be M = $\{P, N\}$. A document is accepted when @ is marked with P when followed by +, with N when followed by − and left unmarked when followed by ?.

An PCMA for such documents can be defined as: $\mathcal{A} = (\Sigma_M, \Sigma_o, Q, 0, \delta, \phi)$ with Σ_M based on $\Sigma = \{@, +, -, ?\}$, $Q = \{0, 1, 2, 3\}$, $\delta = \{\delta(0, @) = 3$, $\delta(0, @_{\underline{P}}) = 1$, $\delta(0, @_{\underline{N}}) = 2$, $\delta(1, +) = 0$, $\delta(2, -) = 0$, $\delta(3, ?) = 0$, $\delta(3, +) = 0$, $\delta(3, -) = 0\}$, and $\phi = \{\phi(0) = accept$, $\phi(1). = reject$, $\phi(2) = reject$, $\phi(3) = reject\}$. This acceptor is represented graphically in FSA 1 (left of Fig. 1). The initial node is indicated with a small arrow, while the accepting node has a circle around it. Note that not every state has a transition for each input symbol. For example, not for the symbol + in state 0. The string is rejected if no transition is defined for the next symbol. Such automata are called incomplete. They can be completed by adding an extra state (a dead state or sink state) with output `reject` and adding a transition to the dead state for each pair (q, s) for which no transition is defined.

FSA 2 shows the CMA version of FSA 1. For the conversion only the transitions that allowed a + or - after an unmarked @ had to be removed. In general this conversion is more complicated, especially for trees. The run of FSA 2 over the correct marking of the sequence @ + @?@− is illustrated in Fig. 1.

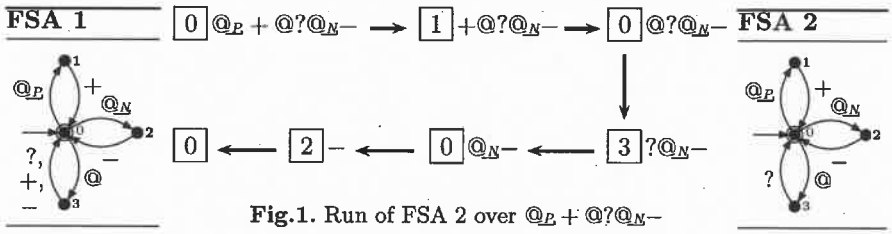


Fig.1. Run of FSA 2 over $@_P + @?@_N-$

A non-deterministic Finite Tree Automaton (NFTA) is a tuple $\mathcal{T} = (\Sigma_i, \Sigma_o, Q, \delta, \phi)$ where Σ_i is a set of input symbols, Σ_o is a set of output symbols, Q is a set of states, ϕ is an output function $Q \rightarrow \Sigma_o$, and δ is a transition relation from the pairs (children states, input symbol) to the next states ($\delta \subseteq (Q^* \times \Sigma_i) \times Q$), such that for each $a \in \Sigma_i$, the set $\{w \in Q^* \mid ((w, a), q) \in \delta\}$ is a regular set of strings over the alphabet Q. A finite tree automaton is deterministic (DFTA) if the transition relation is a function. Unless otherwise indicated we will assume an FTA is deterministic. This definition corresponds to the definition of deterministic bottum-up tree automata in [1], except that we present a practical representation for the transition function.

In practice, we represent the transition function as a set of pairs $(f, \mathcal{A}_f)$ with f an input symbol from Σ_i and $\mathcal{A}_f$ a string automaton $(\Sigma_{if}, \Sigma_{of}, Q_f, q_{0f}, \delta_f, \phi_f)$, where $\Sigma_{if} = \Sigma_{of} = Q$, the set of states from the tree automaton, and where the output function $\phi_f : Q_f \rightarrow Q$ returns the next state for the tree automaton. The tree automaton is deterministic if and only if all string automata associated with the inputs are deterministic.

As was done for string automata, the transition function can be extended into a function $\hat{\delta}$. Let $f(s) \in T(\Sigma_i)$ be a tree with $f \in \Sigma_i$ and $s \in T(\Sigma_i)^*$. Then $\hat{\delta}(f(s)) = \hat{\delta}_f(map(\hat{\delta}, s))$, with $\hat{\delta}_f$ the extended transition function of the string automaton associated with the input symbol f. The function $map(func, seq)$ returns a sequence formed by the results of applying the function *func* on each element of the sequence *seq*.

Example 3. In FTA 1 (left of Fig. 2) we show an example of an FTA. To distinguish them from string automata states, we use $Q_1, Q_2, \ldots$ to refer to tree automaton states. The graphical representation contains in each row an input symbol with its associated FSA. The output function of the FSA's, which returns the next FTA state, is given in the circles that represent the states of the FSA (to allow small circles, we dropped the Q here). The circles are left empty when the output is the dead state of the FTA. The output function for the FTA is indicated by coloring every circle that contains an accepting state. The FSA states are numbered (below the circles) to enable us to refer to them in following sections. We see that $\hat{\delta}_a(1_a, Q_4Q_4) = 2_a$, that $\phi_a(2_a) = Q_5$ and that $\phi(Q_5) = accept$. Fig. 2 illustrates the bottom-up run of FTA 1 on some tree.

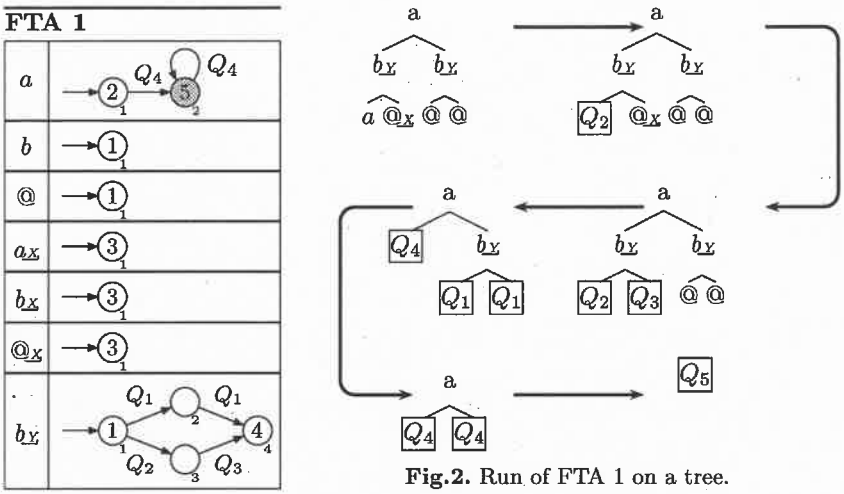


Fig.2. Run of FTA 1 on a tree.

An automaton $\mathcal{A}$ is called a single marking acceptor when it accepts maximally one marking per document. More formally: $\mathcal{A}$ accepts $d \in doc(\Sigma_M) \Rightarrow \forall d' \in (mark_M(strip(d)) \setminus \{d\}) : \mathcal{A}$ rejects d'. By definition, a CMA is always a single marking acceptor.

3 The Single Run Algorithm

Our algorithm uses a CMA. To extract with a CMA, one could feed it all possible markings of the given document, and the single correctly marked version it retains is the solution. In practice though this is not feasible since the number of marked versions of a document equals $(m+1)^n$, with m the number of different markers and n the number of elements in the document. Instead we run it once on a document where each input symbol e is replaced by $mark_M(e)$, the set of all possible markings of the symbol (the input symbol becomes non-deterministic, besides the original symbol it can also be any marking of the original).

Two problems emerge with this approach. First, an acceptor only indicates if a document is accepted or not. So if we pass a non-deterministic marking of a document to the CMA, the result tells us only that there exists a marking for which the document is accepted. Not which specific one. We need a way to run an acceptor such that when it reaches an accepting state, it returns the list of elements that are extracted during the run. We discuss a possible implementation in Section 3.1. A second problem is that given a deterministic CMA and a non-deterministic input, the run will be non-deterministic. In Section 3.2 we show how to solve this. Combining both, we can extract all relevant elements in a single run.

3.1 Keeping track of extractions

In what follows, we assume that a marked element $e_{\underline{X}}$ allows one to identify the original symbol e, the marker X, as well as the position of the symbol in

the document. We modify the states of the automata, we replace a state by a pair (q, E) with q the original state and $E \in \Sigma_M^*$ the sequence of elements to be extracted. More formally, a given FSA $(\Sigma_i, \Sigma_o, Q, q_0, \delta, \phi)$ is redefined as $(\Sigma_i, \Sigma_o', Q', q_0', \delta', \phi')$ where $\Sigma_o' = \Sigma_o \times \Sigma_M^*$, $Q' = Q \times \Sigma_M^*$, $q_0' = (q_0, \epsilon)$. The adapted transition function is $\delta' : Q' \times \Sigma_M \to Q'$ is defined as:

$$\delta'((q, w), e) = \text{ if } e \text{ unmarked then } (\delta(q, e), E) \text{ else } (\delta(q, e), eE)$$

where $q \in Q$, $E \in \Sigma_M^*$, and $e \in \Sigma_M$. Finally, the new output function is defined as $\phi'((q, w)) = (\phi(q), w)$. This modification does not change the actual semantics. The output state of a transition still depends only on the input state and the next symbol. The new list of elements to be extracted only depends on the current list and the next symbol. Fig. 3 shows the same run as in Fig. 1, but with the extracted elements (represented by their positions). The position of each symbol is shown on top of it in the input string. For simplicity, we denote the extracted elements as i_X with i the position and X the marker.

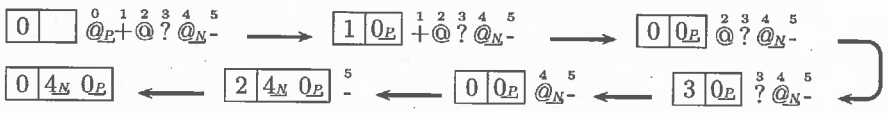


Fig.3. Run of FSA 2 over $@_P + @?@_N-$, keeping extractions.

For FTA's, the situation is a bit more complex. Both the FSA states and the FTA states have sequences of extracted elements linked to them. Both the FTA and the FSA's associated to its input symbols are redefined similar as was done to the FSA's above. For the associated FSA's, q_{0f}' becomes (q_{0f}, ϵ) when f is unmarked, and (q_{0f}, e_X), when $f = e_X$. Further is $\delta_f'((q, E), (p, F)) = (\delta_f(q, p), EF)$, where $E, F \in \Sigma_M^*$, $q \in Q_f$, and $p \in Q$.

This process is illustrated in Fig. 4. This figure shows the same run as in Fig. 2 combined with the element extraction. We have added indices on the nodes of the first tree, such that we can refer to them. In the middle of the figure, the run of the FSA associated with b_Y over the children of node 2 is shown.

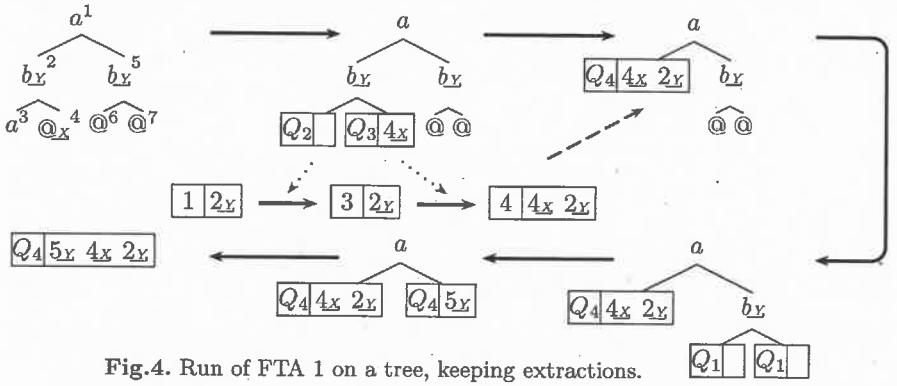


Fig.4. Run of FTA 1 on a tree, keeping extractions.

The memory needed during the run is no longer bounded. Additional memory is needed to keep the sequence, which is the requested answer, in memory. But

since the CMA is a single marking acceptor the size of the sequence cannot exceed the number of symbols in the document. And the number of extracted elements is often much smaller.

3.2 Single Deterministic Run

To cope with the nondeterminism in the input within a single run of the automaton, we use an extension of the standard technique [4] to turn a nondeterministic automaton into a deterministic one.

Given an acceptor $\mathcal{A} = (\Sigma_M, \Sigma_o, Q, q_0, \delta, \phi)$, we create a new acceptor $\tilde{\mathcal{A}} = (\tilde{\Sigma}, \Sigma_o, \tilde{Q}, \tilde{q}_0, \tilde{\delta}, \tilde{\phi})$, in which $\tilde{Q} = 2^Q$, $\tilde{q}_0 = \{q_0\}$. The extension is that instead of keeping the same input alphabet for the new acceptor, the input alphabet is replaced by a nondeterministic alphabet $\tilde{\Sigma} = \{mark_M(e) \mid e \in \Sigma\}$. The new transition relation is also defined on non deterministic input (sets of symbols) instead of the original deterministic input. The definition becomes: $\tilde{\delta}(\tilde{q}, \tilde{e}) = \tilde{q}_n$ with $\tilde{q}_n = \{q_n \in Q \mid \exists \delta(q, e) = q_n$ and $q \in \tilde{q}$ and $e \in \tilde{e}\}$. The output function $\tilde{\phi}$ is defined for every $\tilde{e} \in \tilde{\Sigma}$ as $\tilde{\phi}(\tilde{e}) = accept \Leftrightarrow \exists e \in \tilde{e} : \phi(e) = accept$. To proof that this new automaton is indeed deterministic and accepts exactly the same sequences as the original one, the proof of correctness of the original method [4] needs only some slight extension.

In practice there is no need to create the deterministic automaton explicitly. The transitions can be generated on the fly when needed. To use this automaton for extraction we apply the technique described in Section 3.1. After processing a document $d \in doc(\Sigma_M)$, a final state $\tilde{q} = \hat{\tilde{\delta}}(d)$ is reached ($\hat{\tilde{\delta}}$ is the extended version of $\tilde{\delta}$). Although the run is deterministic, it is possible that the results contains many sequences. This happens when $\tilde{q}$ contains more than one accepting state. Using a single marking acceptor guarantees that there will be maximally one accepting state in $\tilde{q}$. And therefore only a single solution. Fig. 5 finally illustrates the run of our algorithm on the same task as shown in Figures 1 and 3. For clarity we represent the elements of the non-deterministically marked string by their unmarked counterparts. We see that after the third element is processed, the 3 alternative sub-solutions share the same tail. This memory optimization comes natural with the order in which sequences are concatenated in Section 3.1.

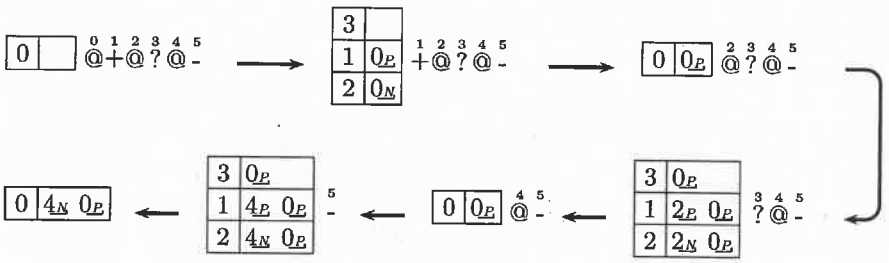


Fig.5. Single deterministic run of FSA 2 over @.+ @?@−

An intuitive explanation, of what the algorithm does, is that it postpones the decision of how an element is marked by considering the different possible markings in parallel. A marking is eliminated as soon as the automaton reaches a reject state for it. When the automaton is a CMA, only a single marking is accepted in the final state.

The approach for FTA's is similar. Only the output function for the associated FSA's will now return sets of FTA states. An example run for an FTA on non deterministic input is given in Fig. 6.

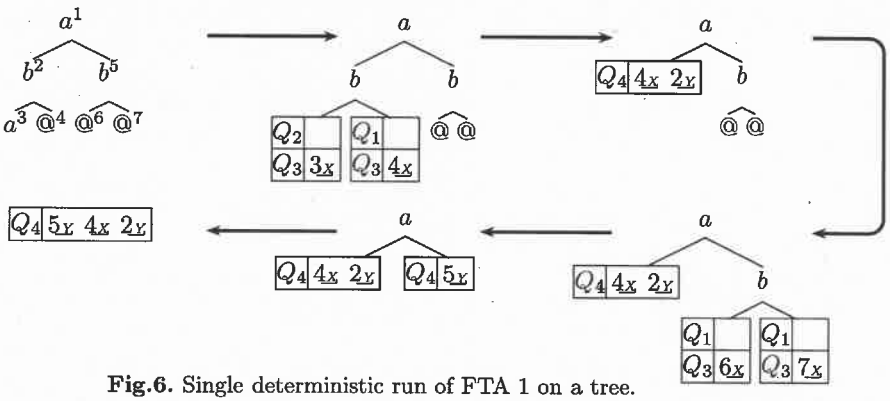


Fig.6. Single deterministic run of FTA 1 on a tree.

4 Complexity

Our algorithm described in Section 3.2 needs to run only once on a document, compared to $\#m \times \#n$ for the algorithm using PCMA's, with $\#m$ the number of markers in M, and $\#n$ the number of elements in the document. However, the complexity of the runs differs, therefore, a better comparison is based on the number of transitions. In the single run algorithm, a single $\tilde{\delta}$-transition consists of multiple δ-transitions, one for each q in a state $\tilde{q}$. We start with the string case. For the algorithm using PCMA's, there is one δ-transition per element of the string, hence, for $\#m \times \#n$ runs, we have $\#\delta = \#m \times \#n^2$ in the worst case. With some optimization this reduces to $\#m \times \#n \times (\#n+1)/2$, still $\mathcal{O}(\#m \times \#n^2)$. For the single run algorithm, $\#\tilde{\delta} = \#n$, but $\#\delta$ is hard to calculate in general. We prove below that $\#\delta$ has an upper-bound that is linear in $\#n$.

The number of δ-transitions in the $\tilde{\delta}$-transition that reaches a certain complex state $\tilde{q}$ depends on the number of simple states, contained in $\tilde{q}$. We proof that for a CMA, every $\tilde{q} \in \tilde{Q}$ contains maximally one occurrence for every $q \in Q$. Suppose a complex state $\tilde{q}$ would contain more than one occurrence of a state q, let's say (q, E_1) and (q, E_2). For every sequence that gets accepted starting from q, there will be two markings accepted. One that has at least the elements of E_1 extracted, and one that has at least the elements of E_2 extracted. A CMA accepts maximally one marking per document. This entails that $E_1{=}E_2$ and therefore $(q, E_1){=}(q, E_2)$, which proofs our assumption. Hence, every complex state $\tilde{q}$ contains maximally $\#q$ (the number of states in Q) simple states. Therefore $\#\delta \leq \#q \times \#n$. This upper-bound can be lowered slightly by taking $\#m$

into account. In Example 4, we give an example of the worst case for $\#m$=1. In this example, $\#\delta = \#q \times \#n - ((\#q - 1) \times \#q)/2$.

Example 4. We introduce following extraction task: given alphabet $\Sigma = \{@\}$ and set of markers $M = \{X\}$, extract the n-th last element of every string. In FSA 3 a solution is given for n=2. The given CMA only accepts those strings that have only the second last element of the string marked with X. Fig. 7 illustrates the run of this CMA over the string @@@@. The extracted element is $2_{\underline{X}}$.

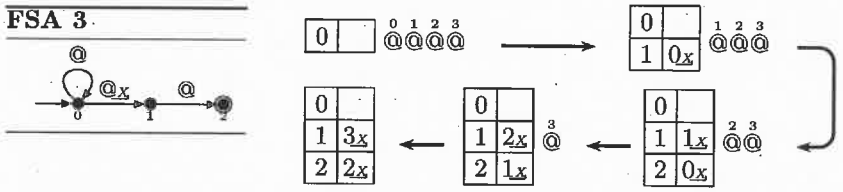


Fig.7. Single deterministic run of FSA on @@@@

In acceptors based on k-local string inference [3, 10], different markers for an element can persist for at most k-steps, because the automaton is not expressive enough to use evidence for some marker that is farther away. In the worst case, this is when every wrong marker is eliminated after k steps, the number of simple states in a complex state is maximally $(\#m+1)^k$. Therefore in the case of k-local automata, the upper-bound becomes $(min((\#m+1)^k, \#q) \times \#n)$.

In the case of trees, the result is similar (a linear upper-bound). We count one $\tilde{\delta}$-transition for each node, and one $\tilde{\delta}_f$-transition for each node except the root node. The number of δ-transitions is smaller than $\#Q \times \#n$, where $\#Q$ is the number of FTA states. The number of auxiliary FSA transitions is smaller than $\#q_{max} \times (\#n - 1)$, where $\#q_{max}$ is the biggest $\#q_f$ over all f in Σ_i.

For both strings and documents we found a linear upper-bound for the number of basic transitions. The factor can be quite large. In practice, different alternative markings do not persist very long and the actual factor is much lower than the upper-bound. See also the experimental results in Section 5.

5 Experiments

We evaluated our algorithm on some data sets from the RISE repository[11]:

- The Bigbook dataset: 235 webpages. We extract the 'name' and the 'address' field. Both fields occur 4299 times in the dataset.
- The IAF dataset: 10 webpages. We extract the 'organization' field, that occurs 94 times, and the 'alt.name' field, that occurs 12 times.
- The Okra dataset: 252 webpages. We extract the 'name' field, 3335 occurrences. The other fields in this dataset are: 'score', date', and 'mail'. Together all these fields amount to 13340 extracted elements.

The experiments were carried out on a Pentium II 400MHz processor, with 128 MB of RAM. The implementation is coded in Java (version 1.4.1). In the

first experiment we used the algorithm from [5] to infer wrappers for the datasets (where needed these wrappers were converted to CMA's [9]). The generated wrappers are all 100% accurate. With these wrappers we extracted the datasets with both the single run algorithm as with the algorithm based on PCMA. The results can be found in Table 1 (each experiment is run at least three times). To give an indication of the size of the pages in the tasks, the last column shows the mean, over all pages, of the number of nodes in the trees.

Table 1. Timings on RISE data sets

data set + field	Single Run (sec.)	PCMA (sec.)	speedup factor	mean #n
bigbook name	3.6 ±0.1	82.3 ±0.1	22.86	505.98
bigbook address	3.3 ±0.1	81.5 ±0.1	24.70	
iaf organization	0.25 ±0.01	3.48 ±0.01	13.92	450.70
iaf alt. name	0.25 ±0.01	3.46 ±0.01	13.84	
okra name	3.3 ±0.1	153.3 ±0.1	46.45	388.38

In a second experiment we used manually crafted wrappers for the Okra dataset. Mainly because the inference algorithm cannot generate wrappers that combine multiple markers. This experiment compares between extraction with $\#m$=1 (name-marker) and $\#m$=4 (all markers). We repeated the experiments for the html-pages represented as trees and as strings (with different wrappers). For strings, $\#n$ means the number of elements in the string. The results are given in Table 2. These results show that $\#m$ has only a small impact on the run-time of our single run algorithm (increases of repectively 1.25 and 1.37 for $\#m$=4). The number of runs of the PCMA increase by 4, yielding a slow-down of respectively 4.65 and 4.35 (due to differences in time needed for extracting the different types of fields). When we compare the first row of Table 2 with the last one of Table 1, we see that we get different timings for the same extraction task. This is due to the fact that the manually crafted wrapper is more specialized towards the trees in the dataset, while the learned one generalizes a bit more. Also the timings for tree automata in Table 2 are better than for string automata. The reason is that FTA's are better suited for structured documents and are therefore less complex than FSA's for the same extraction task.

Table 2. Comparing single-field extraction to multiple-field extraction on OKRA set.

		Single Run (sec.)		PCMA (sec.)		speedup factor	mean #n
tree	name	2.8 ±0.1	× 1.25	138.0 ±0.1	× 4.65	49.29	388.38
	all	3.5 ±0.1		641.5 ±0.1		183.29	
string	name	9.2 ±0.1	× 1.37	344.9 ±0.1	× 4.35	37.49	622.38
	all	12.6 ±0.1		1501.2 ±0.1		119.14	

In a last experiment we took a set of 12 documents from the Okra dataset. We ensured that the number of nodes in these documents were evenly distributed.

We extracted each of these documents a thousand times to get a reasonable estimate for the running time of the Single Run algorithm. The results drawn in Graph 8 show a nice linear behavior.

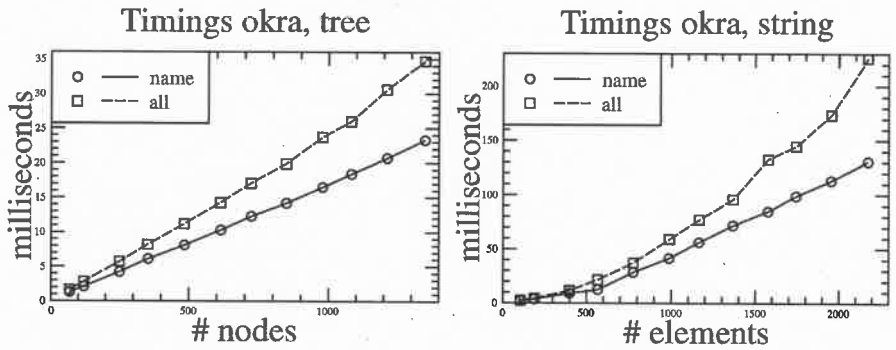


Fig. 8. Experiments illustrating the linear complexity of single run extraction.

6 Extracting Internal Nodes

In the extraction task of the previous section, the extracted elements are always leaf nodes in the HTML-tree. To the algorithm, the only difference between a leaf node and an internal node is the number of children. The algorithm is therefore perfectly capable of extracting internal nodes when it is given a CMA that accepts trees in which internal nodes are marked. An example of this is FTA 1. We see that some internal nodes are extracted in Fig. 6.

A first application hereof is to expand the wrapper induction to structured values. This could be a first step in extracting tuples of fields. Or to extract a text value, that has some parts in bold or otherwise accentuated. Or to actually extract structured pieces of the document.

A second application could be the counting of substructures, which is used in tree mining. Instead of learning a wrapper, a CMA can be generated to mark all subtrees in a tree, that adhere to some description. After running the algorithm, we only have to count the number of extracted elements. These descriptions could be acceptors (for unmarked trees), one acceptor for every pattern we want to count. Patterns should be disjunct, otherwise they should be split until they are. A subtree belongs to the requested pattern when it is accepted by the acceptor. A CMA can then be created automatically, that marks and extracts the root of a pattern (with a mark identifying the pattern). Since the marking of a node depends only on the tree of which it is a root, the decision to mark is never postponed, leading to only $2\#n$-1 simple transitions for any number of patterns.

A description of a pattern could be more expressive though. We could count subtrees that belong to some subset of trees, but only those for which some specific arrangement of siblings and parent(s) hold. In this case the marking of a node will stay indetermined until definitive proof is found in its surrounding.

7 Conclusion

We have presented a new, faster technique to extract information from structured documents, using either string or tree automata. This technique uses CMA's instead of PCMA's. The former accepts less markings, and will therefore reduce the number of intermediate transitions earlier in the run. Both complexity analysis and experiments indicate a linear behavior with regard to document sizes.

We suggested an alternative application for this technique, namely counting occurrences of substructures. More research is needed to study the viability of this application.

References

1. Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical Report HKUST-TCSC-2001-05.
2. Boris Chidlovskii, Jon Ragetli, and Maarten de Rijke. Wrapper generation via grammar induction. In *Proc. 11th European Conference on Machine Learning (ECML)*, volume 1810, pages 96–108. Springer, Berlin, 2000.
3. Pedro García and Enrique Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.
4. Arthur Gill. *Applied Algebra for the Computer Sciences.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1976.
5. Raymondus Kosala, Maurice Bruynooghe, Hendrik Blockeel, and Jan Van den Bussche. Information extraction from web documents based on local unranked tree automaton inference. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 403–408. Morgan Kaufmann, 2003.
6. Raymondus Kosala, Jan Van den Bussche, Maurice Bruynooghe, and Hendrik Blockeel. Information extraction in structured documents using tree automata induction. In *PKDD*, volume 2431 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2002.
7. Nickolas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.
8. Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.
9. Stefan Raeymaekers. A conversion algorithm between pcma and cma. draft.
10. Juan Ramón Rico-Juan, Jorge Calera-Rubio, and Rafael C. Carrasco. Probabilistic k-testable treelanguages. In A.L. Oliveira, editor, *Proceedings of 5th International Colloquium, ICGI*, pages 221–228, 2000.
11. Rise (1998). a repository of online information sources used in information extraction tasks. [http://www.isi.edu/info-agents/RISE/index.html]. University of Southern California, Information Sciences Institute.
12. Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.